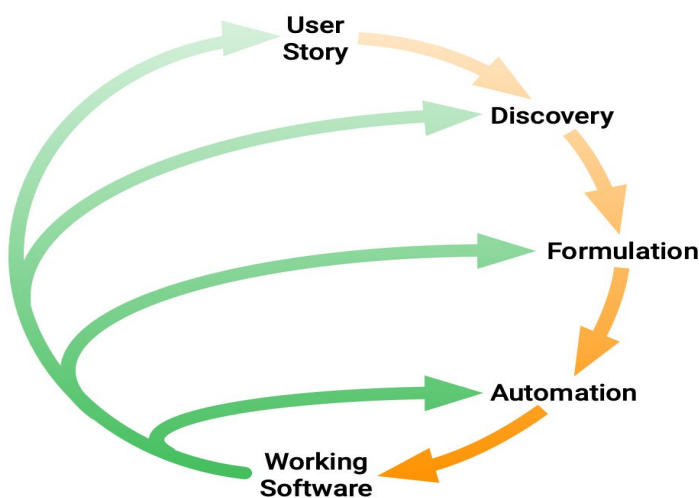


CucumberJs

Cucumber is a tool that supports Behaviour-Driven Development(BDD). BDD attempts to help software teams in closing the gap between business people and technical people. It is a process for the team where the team collaborates to create a natural language description of the behaviour of the software which is connected to automated tests to produce living documentation. BDD activity is essentially a three step, iterative process:

1. A user story i.e. a small upcoming change in the system is taken and examples of the new functionality are taken.
2. Those examples are documented in a way that can be automated.
3. The behaviour described by each documented example is implemented starting with an automated test.



The idea is to make each change small and iterate rapidly, moving back up a level each time you need more information. Each time you automate and implement a new example, you've added something valuable to your system, and you're ready to respond to feedback.

We call these practices *Discovery*, *Formulation*, and *Automation*.

- Discovery: What it could do
- Formulation: What it should do
- Automation: What it actually does

Cucumber reads executable specifications written in plain text and validates that the software does what those specifications say. The specifications consist of multiple examples, or scenarios.

Each scenario is a list of steps for Cucumber to work through. Cucumber verifies that the software conforms with the specification and generates a report indicating success or failure for each scenario.

In order for Cucumber to understand the scenarios, they must follow some basic syntax rules, called Gherkin.

Drivers of BDD in cucumber:

As the word 'driver' suggests, these items are the main parts of a cucumber.js framework, and are the part where the programmer/tester interacts with the program. It contains 2 files which are defined by a programmer/tester, to help him/her test the working of an api. These files are:

- .feature file
- .js file

Gherkin

- ❖ Gherkin uses a set of special [keywords](#) to give structure and meaning to executable specifications.
- ❖ Or Gherkin is a plain English text language, which helps the tool - Cucumber to interpret and execute the test scripts.
- ❖ Most lines in a Gherkin document start with one of the [keywords](#).

Keywords

The primary keywords are:

- Feature
- Scenario
- Given, When Then , And, But for steps
- Scenario Outline
- Background

There are a few secondary keywords as well:

- " " (Doc Strings)
- | (Data Tables)
- @ (Tags)
- # (Comments)

Features

- ❖ The purpose of the Feature keyword is to provide a high-level description of a software feature, and to group related scenarios.
- ❖ A Feature keyword can be defined as a standalone unit or functionality of a project.
- ❖ Each Gherkin file begins with the Feature keyword. It just gives you a convenient place to put some summary documentation about the group of tests that follow.

Here's an example:

Feature: Google Search

The User should be able to search from the google search bar

- ❖ The file, in which Cucumber tests are written, is known as feature files. It is advisable that there should be a separate feature file, for each feature under test. The extension of the feature file needs to be “.feature”.

Scenario

- ❖ Scenario is one of the core Gherkin structures. Every scenario starts with the keyword “Scenario:” (or localized one) and is followed by an optional scenario title. Each feature can have one or more scenarios and every scenario consists of one or more steps.
- ❖ To actually express the behavior we want, each feature contains several scenarios. Each scenario is a single concrete example of how the system should behave in a particular situation.
- ❖ When Cucumber runs a scenario, if the system behaves as described in the scenario, then the scenario will pass; if not, it will fail. Each time you add a new scenario to your Cucumber test suite and make it pass.
- ❖ Each scenario tells a little story describing something that the system should be able to do.

Give, When, Then

Given -

- It describes the pre-requisite for the test to be executed.
- Example – **GIVEN** I have \$100 in my account

When -

- It defines the trigger point for any test scenario execution.
- Example – **WHEN** I request \$20

Then -

- Then holds the expected result for the test to be executed.
- Example – **THEN** \$20 should be dispensed

Example : -

Scenario: Successful withdrawal from an account in credit

Given I have \$100 in my account # the context

When I request \$20 # the event(s)

Then \$20 should be dispensed # the outcome(s)

So, we use Given to set up the context where the scenario happens, When to interact with the system somehow, and Then to check that the outcome of that interaction was what we expected.

And, But

Each of the lines in a scenario is known as a step. We can add more steps to each Given, When, or Then section of the scenario using the keywords And and But:

And -

- It provides the logical AND condition between any two statements. AND can be used in conjunction with GIVEN, WHEN and THEN statement.
- Example - WHEN I enter my "<username>" AND I enter my "<password>"

But -

- It signifies logical OR condition between any two statements. OR can be used in conjunction with GIVEN, WHEN and THEN statement.
- Example - THEN login should be successful. BUT home page should not be missing.

Example : -

Scenario: Attempt withdrawal using stolen card

Given I have \$100 in my account

But my card is invalid

When I request \$50

Then my card should not be returned

And I should be told to contact the bank

Scenario Outline

- ❖ The Scenario Outline keyword can be used to run the same Scenario multiple times, with different combinations of values.
- ❖ Scenario outline basically replaces variable/keywords with the value from the table. Each row in the table is considered to be a scenario.
- ❖ The keyword **Scenario Template** is a synonym of the keyword **Scenario Outline**.
- ❖ For Example : -

Scenario: eat 5 out of 12

Given there are 12 cucumbers

When I eat 5 cucumbers

Then I should have 7 cucumbers

Scenario: eat 5 out of 20
 Given there are 20 cucumbers
 When I eat 5 cucumbers
 Then I should have 15 cucumbers

- ❖ We can collapse these two similar scenarios into a Scenario Outline.
- ❖ When we define any scenario with scenario outline, we can specify one test scenario and at the bottom of it we can provide a number of inputs. The scenario will get executed as many times as the number of inputs provided.

Scenario Outline: eating
 Given there are <start> cucumbers
 When I eat <eat> cucumbers
 Then I should have <left> cucumbers

Examples:

	start		eat		left	
	12		5		7	
	20		5		15	

- ❖ A **Scenario Outline** must contain an **Examples** (or **Scenarios**) section. Its steps are interpreted as a template which is never directly run. Instead, the **Scenario Outline** is run once for each row in the **Examples** section beneath it (not counting the first header row).
- ❖ The steps can use **<>** delimited parameters that reference headers in the examples table. Cucumber will replace these parameters with values from the table before it tries to match the step against a step definition.

Background

- ❖ **Background in Cucumber** is used to define a step or series of steps that are common to all the tests in the feature file. It allows you to add some context to the scenarios for a feature where it is defined.
- ❖ A Background is much like a scenario containing a number of steps. But it runs before each and every scenario for a feature in which it is defined.

Doc String

- ❖ Doc Strings are used for a larger piece of text to a step definition.
- ❖ Example :-

```
Given a blog post named "Random" with Markdown body
"""
Some Title, Eh?
=====
Here is the first paragraph of my blog post. Lorem ipsum dolor
sit amet,
consectetur adipiscing elit.
"""
```

Data tables

- ❖ Data tables are used for the defining the list of values in table
- ❖ Example -

```
Given the following users exist:
| name   | email
| Aman   | aman@gmail.com
| Jatin  | jatin@gmail.com
| Ravi   | ravi@gmail.com
```

Feature file:

A Feature file contains a high level description of the Test Scenario in simple language. A feature file would be the one containing the gherkin steps. This file has an extension of '.feature'. A feature file contains the following components:

- **Feature:** A feature would describe the current test script which has to be executed, which means it would describe for what feature this script has been written in either a single line or a descriptive one in multi-line.
- **Scenario:** Scenario describes the steps and expected outcome for a particular test case. In other words, like a normal scenario, it defines one of the instances the API would undergo. It contains "GIVEN WHEN THEN" that defines various scenarios.
- **Scenario Outline:** Same scenario can be executed for multiple sets of data using scenario outline. The data is provided by a tabular structure separated by (|)
- **Given:** It specifies the context of the text to be executed. By using datatables "Given", step can also be parameterized.
- **When:** "When" specifies the test action that has to performed.
- **Then:** The expected outcome of the test can be represented by "Then".
- **Background:** A Background allows you to add some context to the scenarios that follow it. It can contain one or more Given steps, which are run before each scenario, but after any Before hooks. A Background is placed before the first Scenario/Example, at the same level of indentation.

The syntax of a basic feature file would be:

Feature: (Description of a feature)....

Scenario: (Description of a scenario)....

Given (The state the system is currently in)....

When (Action expected to change the state of system)....

Then (The state the system must be after executing the action)

You can specify additional details by adding various steps in the files:

Feature: (Description of a feature)....

.... (Additional description)....

Background:

Given (The state the system is currently in)....

And (Some other state).....

..

Scenario: (Description of a scenario)....

Given (The state the system is currently in)....

And (Some other state).....

And (Some other state).....

..

When (Action expected to change the state of system)....

And (Some other action).....

..

OR

...(Pass a table with values)...

Then (The state the system must be after executing the action)

But (Anything else that needs to be verified)....

Scenario outline: (Description of a scenario)....

Given (The state <var_name> the system is currently in)....

When (Action <var_name> expected to change the state of system)....

Then (State <var_name> the system must be after executing the action)

Examples:

var1	var2	var3
100	5	105
99	1234	1333
12	5	18

Example:

Feature: Simple maths

In order to do maths

As a developer

I want to increment variables

Scenario: easy maths

Given a variable set to 1

When I increment the variable by 1

Then the variable should contain 2

Scenario Outline: much more complex stuff

Given a variable set to <var>

When I increment the variable by <increment>

Then the variable should contain <result>

Examples:

var	increment	result
-----	-----------	--------

100	5	105
-----	---	-----

99	1234	1333
----	------	------

12	5	17
----	---	----

Step-definition file:

The gherkin steps written in pure English text wouldn't be of any meaning to the system, it needs some code that can be executed or mapped to the steps. This is what the step-definitions file contains, the javascript code that maps to the gherkin steps written in the feature file. When Cucumber executes a Gherkin step in a scenario, it will look for a matching step definition to execute.

Steps Definition file contains the actual code to execute the Test Scenario in the Features file. Cucumber supports 2 types of execution for step-definitions:

1. Cucumber expressions
2. Regular expressions

Basic syntax of a step definitions file:

```
Import {Given, When, Then} from 'cucumber';
```

OR

```
var setWorldConstructor = Cucumber.setWorldConstructor;
```

```
var Given = Cucumber.Given;
```

```
var When = Cucumber.When;
```

```
var Then = Cucumber.Then;
```

```
//other import statements
```

```
Given('... (The state the system is currently in)...', function( ...args...){
```

```
    // code
```

```
});
```

```
When('... (Action expected to change state of system)...', function( ...args...){
```

```
    // code
```

```
});
```

```
Then('... (State the system must be after executing action) ...', function( ...args...){
```

```
    // code
```

```
});
```

Example:

```
var setWorldConstructor = Cucumber.setWorldConstructor;
```

```
var Given = Cucumber.Given;
```

```
var When = Cucumber.When;
```

```
var Then = Cucumber.Then;
```

```
var expect = chai.expect;
```

```
///// World /////
```

```
// Call 'setWorldConstructor' with to your custom world (optional)
```

```
var CustomWorld = function() {
```

```
    this.variable = 0;
```

```

};

CustomWorld.prototype.setTo = function(number) {
  this.variable = parseInt(number);
};

CustomWorld.prototype.incrementBy = function(number) {
  this.variable += parseInt(number);
};

setWorldConstructor(CustomWorld);

///// Step definitions /////
// use 'Given', 'When' and 'Then' to declare step definitions

Given('a variable set to {int}', function(number) {
  this.setTo(number);
});

When('I increment the variable by {int}', function(number) {
  this.incrementBy(number);
});

Then('the variable should contain {int}', function(number) {
  expect(this.variable).toEqual(number);
});

```

CUCUMBER ON THE GO:

Feature	Step Definitions	Output
1	Feature: Simple maths	
2	In order to do maths	
3	As a developer	
4	I want to increment variables	
5		
6	Scenario: easy maths	
7	Given a variable set to 1	
8	When I increment the variable by 1	
9	Then the variable should contain 2	
10		
11	Scenario Outline: much more complex stuff	
12	Given a variable set to <var>	
13	When I increment the variable by <increment>	
14	Then the variable should contain <result>	
15		
16	Examples:	
17	var increment result	
18	100 5 105	
19	99 1234 1333	
20	12 5 18	

Feature file



```
1 // Cucumber and chai have been loaded in the browser
2 var setWorldConstructor = Cucumber.setWorldConstructor;
3 var Given = Cucumber.Given;
4 var When = Cucumber.When;
5 var Then = Cucumber.Then;
6 var expect = chai.expect;
7
8 ///// World /////
9 //
10 // Call 'setWorldConstructor' with to your custom world (optional)
11 //
12
13 - var CustomWorld = function() {
14   this.variable = 0;
15 };
16
17 - CustomWorld.prototype.setTo = function(number) {
18   this.variable = parseInt(number);
19 };
20
21 - CustomWorld.prototype.incrementBy = function(number) {
22   this.variable += parseInt(number);
23 };
24
25 setWorldConstructor(CustomWorld);
26
27 ///// Step definitions /////
28 //
29 // use 'Given', 'When' and 'Then' to declare step definitions
30 --
```

Step-definitions file



```
.....
4 scenarios (4 passed)
12 steps (12 passed)
0m00.011s

Exit Status: 0
```

Output of the test

Apache Kafka

- Kafka is written in Scala and Java. Apache Kafka is a publish-subscribe based fault-tolerant messaging system. It is fast, scalable and distributed by design.
- This technology is used in big data because in big data enormous volume of data is used.
- Because of data, we have two main challenges
 - First is how to collect a large volume of data.
 - Second is analyse the collected data.

- To overcome those challenges we must need a messaging system.
- Kafka is designed for distributed high throughput systems.
- Is an alternative of traditional broker and comparison to other messaging systems.
- Kafka has better throughput built-in partitioning, replication and inherent fault-tolerance, which makes it a good fit for large-scale message processing application.
- Kafka is suitable for both online and offline message consumption.
- In Kafka message are persisted on the disk and replicated within the cluster to prevent data loss.
- Kafka is integrated very well with apache storm and spark for real-time streaming data analysis

Benefits of Kafka

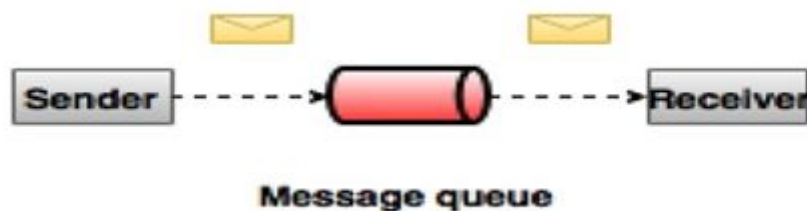
- Reliability- Kafka is distributed, partitioned, replicated and fault tolerance.
- Scalability- Kafka message system scales easily without downtime.
- Durability- Kafka uses “distributed commit log” which means message persists on disk as fast as possible, hence it is durable.
- Performance- Kafka has high throughput for both publishing and subscribing messages. It maintains stable performance even when many TB of messages are stored.

What is a messaging system?

- Messaging system is responsible for transferring data from one application to another, so the application can focus on data but not worry about how to share it.
- Messages are queued asynchronously between the client application and messaging system.
- Two types of messaging patterns are available
 - Point to point
 - Publish-subscribe (pub-sub) messaging system mostly followed.

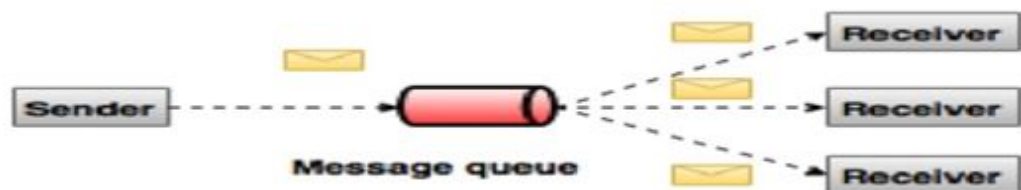
Point to point messaging system

- In this system, messages are persisted in a queue
- One or more consumer can consume the message in the queue but a particular message can be consumed by a maximum of one consumer only.
- When a consumer reads a message in the queue it disappears from that queue
- For example order processing system where each order will be processed by one order processor.



Publish-Subscribe messaging system (pub-sub)

- Pub-Sub system message is persisted in a topic.
- Unlike point to point system consumer can subscribe to one or more topic and consume all the message on that topic.
- In a pub-sub system, message producer is called publishers and the message consumer is called subscribers.
- For example, Dish Tv which publishes different channels like sport, movie, music, etc anyone can subscribe to their own set of channels and get them whenever their subscribed channels are available.



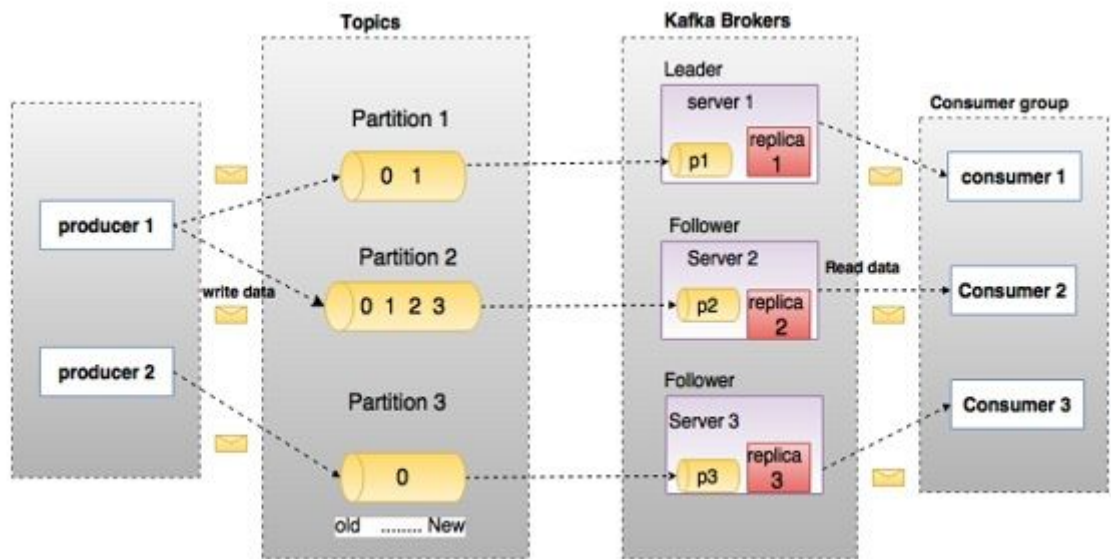
Need of Kafka

- Kafka is a unified platform for handling all the real-time data feeds.
- Its supports low latency message delivery and give guarantee for fault tolerance in the presence of machine failures.

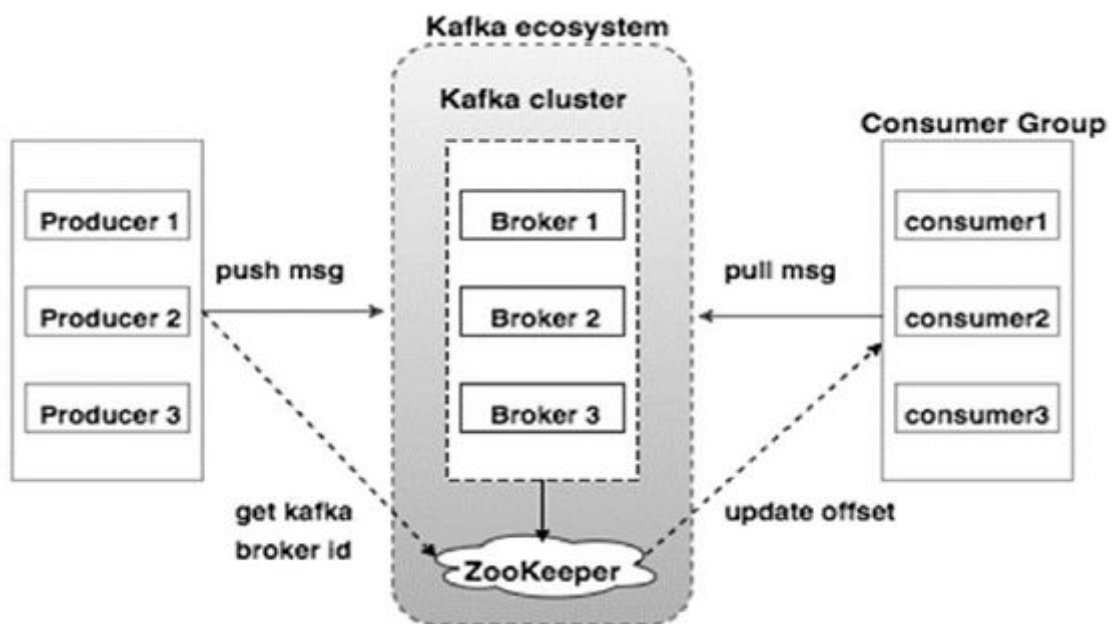
- Kafka is very fast, performs 2 million writes/sec.
- Whatever data we write it goes to the page cache of the OS (RAM). This makes it very efficient to transfer data from page cache to a network socket.

Main Terminologies used in Kafka

- **Topics**- A stream of messages belonging to a particular category is called a topic. Data is stored in topics.
- **Partition**- Topics may have many partitions, so it can handle an arbitrary amount of data.
- **Partition offset**- Each partitioned message has a unique sequence id called as offset
- **Replicas of partition**- Replicas are nothing but backups of a partition. Replicas are never read or write data. They are used to prevent data loss.
- **Brokers**- Brokers are simple system responsible for maintaining the published data.
 - Each broker may have zero or more partitions per topic. Assume, if there are N partitions in a topic and N number of brokers, each broker will have one partition.
 - Assume if there are N partitions in a topic and more than N brokers ($n + m$), the first N broker will have one partition and the next M broker will not have any partition for that particular topic.
 - Assume if there are N partitions in a topic and less than N brokers ($n - m$), each broker will have one or more partition sharing among them. This scenario is not recommended due to unequal load distribution among the broker.
- **Producers**- Producers are the publisher of messages to one or more Kafka topics. Producers send data to Kafka brokers. Every time a producer publishes a message to a broker, the broker simply appends the message to the last segment file. Actually, the message will be appended to a partition. A producer can also send messages to a partition of their choice.
- **Consumers**- Consumers read data from brokers. Consumers subscribe to one or more topics and consume published messages by pulling data from the brokers.
- **Leader**- Leader is the node responsible for all reads and writes for the given partition. Every partition has one server acting as a leader.
- **Follower**- Node which follows leader instructions are called as a follower. If the leader fails, one of the followers will automatically become the new leader. A follower acts as a normal consumer, pulls messages and updates its own data store.



Cluster architecture



- **Broker-** Kafka cluster typically consists of multiple brokers to maintain load balance. Kafka brokers are stateless, so they use ZooKeeper for maintaining their cluster state. One Kafka broker instance can handle hundreds of

thousands of reads and writes per second and each broker can handle TB of messages without performance impact. Kafka broker leader election can be done by ZooKeeper

- **ZooKeeper-** ZooKeeper is used for managing and coordinating Kafka broker. ZooKeeper service is mainly used to notify producer and consumer about the presence of any new broker in the Kafka system or failure of the broker in the Kafka system. As per the notification received by the Zookeeper regarding presence or failure of the broker then producer and consumer take the decision and starts coordinating their task with some other broker.
- **Producers-** Producers push data to brokers. When the new broker is started, all the producers search it and automatically sends a message to that new broker. Kafka producer doesn't wait for acknowledgements from the broker and sends messages as fast as the broker can handle.
- **Consumers-** Since Kafka brokers are stateless, which means that the consumer has to maintain how many messages have been consumed by using partition offset. If the consumer acknowledges a particular message offset, it implies that the consumer has consumed all prior messages. The consumer issues an asynchronous pull request to the broker to have a buffer of bytes ready to consume. The consumers can rewind or skip to any point in a partition simply by supplying an offset value. A consumer offset value is notified by ZooKeeper.

Workflow

- Kafka is simply a collection of topics split into one or more partitions. A Kafka partition is a linearly ordered sequence of messages, where each message is identified by their index (called as offset).
- In Kafka, Incoming messages are written at the end of a partition and messages are sequentially read by consumers. Durability is provided by replicating messages to different brokers.
- Kafka provides both pub-sub and queue-based messaging system in a fast, reliable, persisted, fault-tolerance and zero downtime manner. In both cases, producers simply send the message to a topic and consumer can choose any one type of messaging system depending on their need.

Workflow of pub-sub messaging

- Producers send message to a topic at regular intervals.
- Kafka broker stores all messages in the partitions configured for that particular topic. It ensures the messages are equally shared between partitions. If the producer sends two messages and there are two partitions, Kafka will store

one message in the first partition and the second message in the second partition.

- Consumer subscribes to a specific topic.
- Once the consumer subscribes to a topic, Kafka will provide the current offset of the topic to the consumer and also saves the offset in the Zookeeper ensemble.
- Consumer will request the Kafka in a regular interval (like 100 Ms) for new messages.
- Once Kafka receives the messages from producers, it forwards these messages to the consumers.
- Consumer will receive the message and process it.
- Once the messages are processed, the consumer will send an acknowledgement to the Kafka broker.
- Once Kafka receives an acknowledgement, it changes the offset to the new value and updates it in the Zookeeper. Since offsets are maintained in the Zookeeper, the consumer can read next message correctly even during server outages.
- This above flow will repeat until the consumer stops the request.
- Consumer has the option to rewind/skip to the desired offset of a topic at any time and read all the subsequent messages.

Workflow of queue messaging / consumer Group

- In a queue messaging system instead of a single consumer, a group of consumers having the same Group ID will subscribe to a topic. In simple terms, consumers subscribing to a topic with the same Group ID are considered as a single group and the messages are shared among them. Let us check the actual workflow of this system.
- Producers send message to a topic in a regular interval.
- Kafka stores all messages in the partitions configured for that particular topic similar to the earlier scenario.
- A single consumer subscribes to a specific topic, assume Topic-01 with Group ID as Group-1.
- Kafka interacts with the consumer in the same way as Pub-Sub Messaging until new consumer subscribes the same topic, Topic-01 with the same Group ID as Group-1.
- Once the new consumer arrives, Kafka switches its operation to share mode and shares the data between the two consumers. This sharing will go on until the number of consumers reach the number of partition configured for that particular topic.
- Once the number of consumers exceeds the number of partitions, the new consumer will not receive any further message until any one of the existing

consumers unsubscribes. This scenario arises because each consumer in Kafka will be assigned a minimum of one partition and once all the partitions are assigned to the existing consumers, the new consumers will have to wait.

- This feature is also called as Consumer Group. In the same way, Kafka will provide the best of both the systems in a very simple and efficient manner.

Kafka Examples.

- Kafka uses [ZooKeeper](#) so you need to first start a ZooKeeper server.
- Kafka uses Zookeeper for the following:
- **Electing a controller.** The controller is one of the brokers and is responsible for maintaining the leader/follower relationship for all the partitions. When a node shuts down, it is the controller that tells other replicas to become partition leaders to replace the partition leaders on the node that is going away. Zookeeper is used to elect a controller, make sure there is only one and elect a new one if it crashes.
- **Cluster membership** - which brokers are alive and part of the cluster? This is also managed through ZooKeeper.
- **Topic configuration** - which topics exist, how many partitions each has, where are the replicas, who is the preferred leader, what configuration overrides are set for each topic
- **Quotas** - how much data is each client allowed to read and write
- **ACLs** - who is allowed to read and write to which topic
- **Consumer Management** - Which consumer groups exist, who are their members and what is the latest offset each group got from each partition.

To start a zookeeper server:

> bin/zookeeper-server-start.sh config/zookeeper.properties

```
dhooop@dhooop-ip510: ~/Downloads/kafka/kafka_2.12-2.4.1
File Edit View Search Terminal Help

dhooop@dhooop-ip510:~/Downloads/kafka/kafka_2.12-2.4.1$ bin/zookeeper-server-start
.sh config/zookeeper.properties
[2020-04-06 11:22:10,316] INFO Reading configuration from: config/zookeeper.prop
erties (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2020-04-06 11:22:10,326] WARN config/zookeeper.properties is relative. Prepend
./ to indicate that you're sure! (org.apache.zookeeper.server.quorum.QuorumPeerC
onfig)
[2020-04-06 11:22:10,338] INFO clientPortAddress is 0.0.0.0:2181 (org.apache.zoo
keeper.server.quorum.QuorumPeerConfig)
[2020-04-06 11:22:10,338] INFO secureClientPort is not set (org.apache.zookeeper
.server.quorum.QuorumPeerConfig)
[2020-04-06 11:22:10,369] INFO autopurge.snapRetainCount set to 3 (org.apache.zo
ookeeper.server.DataDirCleanupManager)
[2020-04-06 11:22:10,369] INFO autopurge.purgeInterval set to 0 (org.apache.zook
eeper.server.DataDirCleanupManager)
[2020-04-06 11:22:10,369] INFO Purge task is not scheduled. (org.apache.zookeep
er.server.DataDirCleanupManager)
[2020-04-06 11:22:10,369] WARN Either no config or no quorum defined in config,
running in standalone mode (org.apache.zookeeper.server.quorum.QuorumPeerMain)
[2020-04-06 11:22:10,371] INFO Log4j found with jmx enabled. (org.apache.zookeep
er.jmx.ManagedUtil)
[2020-04-06 11:22:10,480] INFO Reading configuration from: config/zookeeper.prop
erties (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2020-04-06 11:22:10,481] WARN config/zookeeper.properties is relative. Prepend
```

Now start the Kafka server:

> bin/kafka-server-start.sh config/server.properties

```
dhooop@dhooop-ip510: ~/Downloads/kafka/kafka_2.12-2.4.1
File Edit View Search Terminal Tabs Help

dhooop@dhooop-ip510:~/Downloads/kafka/kafka_2.12-2.4.1$ bin/kafka-server-start.sh
config/server.properties
[2020-04-06 11:30:24,718] INFO Registered kafka:type=kafka.Log4jController MBean
(kafka.utils.Log4jControllerRegistration$)
[2020-04-06 11:30:26,159] INFO Registered signal handlers for TERM, INT, HUP (or
g.apache.kafka.common.utils.LoggingSignalHandler)
[2020-04-06 11:30:26,162] INFO starting (kafka.server.KafkaServer)
[2020-04-06 11:30:26,167] INFO Connecting to zookeeper on localhost:2181 (kafka.
server.KafkaServer)
[2020-04-06 11:30:26,326] INFO [ZooKeeperClient Kafka server] Initializing a new
session to localhost:2181. (kafka.zookeeper.ZooKeeperClient)
[2020-04-06 11:30:26,349] INFO Client environment:zookeeper.version=3.5.7-f0fdd5
2973d373ffd9c86b81d99842dc2c7f660e, built on 02/10/2020 11:30 GMT (org.apache.zo
ookeeper.ZooKeeper)
[2020-04-06 11:30:26,349] INFO Client environment:host.name=dhooop-ip510 (org.apa
che.zookeeper.ZooKeeper)
[2020-04-06 11:30:26,349] INFO Client environment:java.version=11.0.6 (org.apach
e.zookeeper.ZooKeeper)
[2020-04-06 11:30:26,349] INFO Client environment:java.vendor=Ubuntu (org.apach
e.zookeeper.ZooKeeper)
[2020-04-06 11:30:26,349] INFO Client environment:java.home=/usr/lib/jvm/java-11
-openjdk-amd64 (org.apache.zookeeper.ZooKeeper)
```

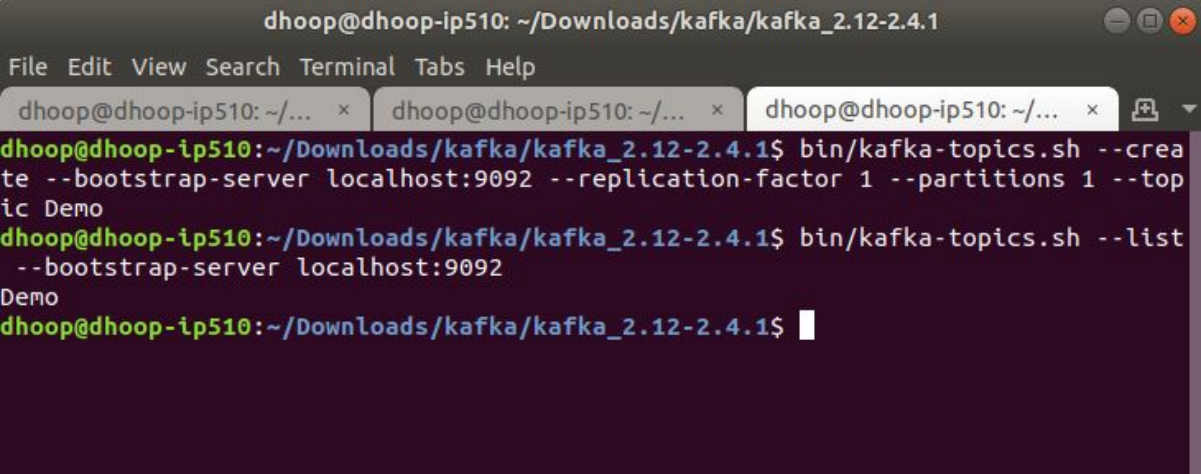
Create a topic:

Using following command we can create a topic named "Demo" with a single partition and only one replica:

```
> bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic Demo
```

We can see the list of topics using following command:

```
> bin/kafka-topics.sh --list --bootstrap-server localhost:9092
```

A screenshot of a terminal window titled 'dhoop@dhoop-ip510: ~/Downloads/kafka/kafka_2.12-2.4.1'. The terminal shows the execution of two Kafka commands. The first command is 'bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic Demo', which creates a topic named 'Demo'. The second command is 'bin/kafka-topics.sh --list --bootstrap-server localhost:9092', which lists the topics and outputs 'Demo'. The terminal has a dark background with green and white text. There are three tabs open at the top, all showing the same path. The window has standard Linux window controls (minimize, maximize, close) in the top right corner.

```
dhoop@dhoop-ip510: ~/Downloads/kafka/kafka_2.12-2.4.1
File Edit View Search Terminal Tabs Help
dhoop@dhoop-ip510: ~/... x dhoop@dhoop-ip510: ~/... x dhoop@dhoop-ip510: ~/... x
dhoop@dhoop-ip510:~/Downloads/kafka/kafka_2.12-2.4.1$ bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic Demo
dhoop@dhoop-ip510:~/Downloads/kafka/kafka_2.12-2.4.1$ bin/kafka-topics.sh --list --bootstrap-server localhost:9092
Demo
dhoop@dhoop-ip510:~/Downloads/kafka/kafka_2.12-2.4.1$
```

Kafka comes with a command line client that will take input from a file or from standard input and send it out as messages to the Kafka cluster. By default, each line will be sent as a separate message.

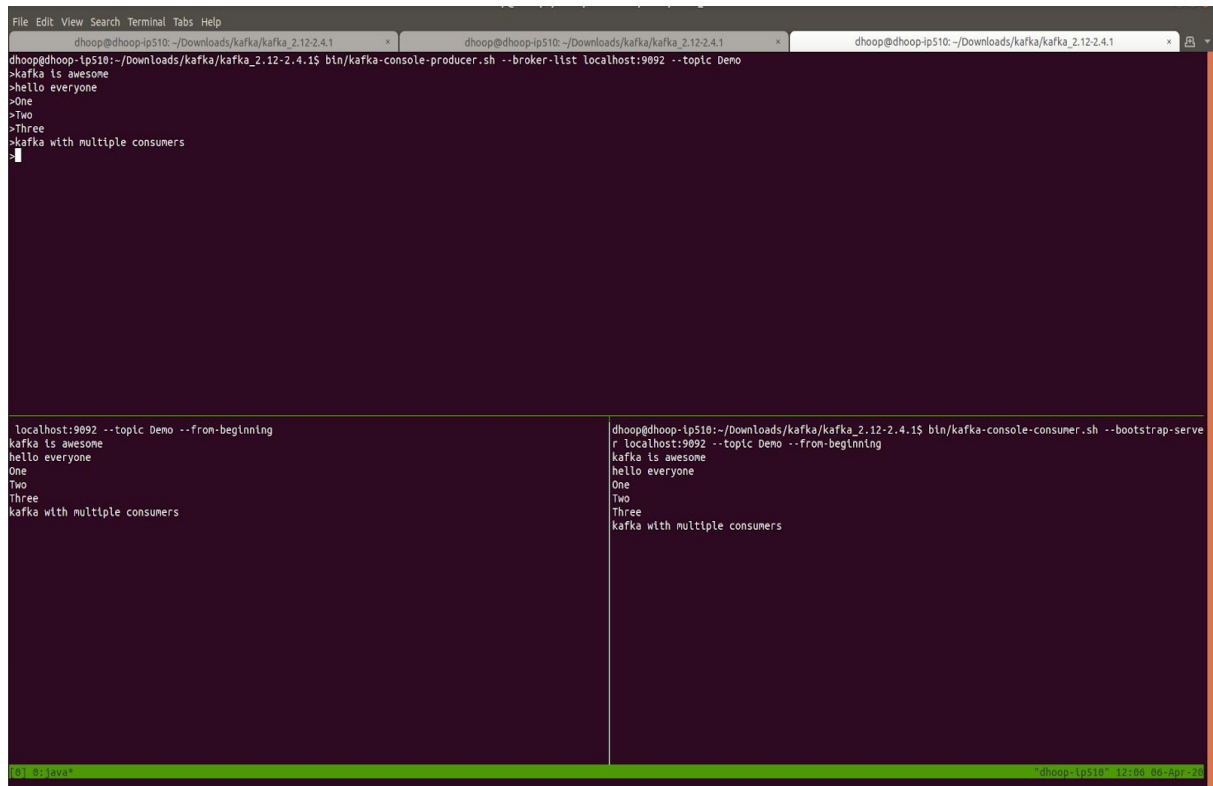
Run the producer and then type a few messages into the console to send to the server.

Write a file to the kafka producer.

```
> bin/kafka-console-producer.sh --broker-list localhost:9092 --topic Demo < ~/Desktop/GeoLocationData.txt
```

Start producer in console.

```
> bin/kafka-console-producer.sh --broker-list localhost:9092 --topic Demo
```



The screenshot shows a terminal window with three tabs. The active tab is titled 'dhoop@dhoop-ip510: ~/Downloads/kafka/kafka_2.12-2.4.1'. The terminal content is as follows:

```
dhoop@dhoop-ip510:~/Downloads/kafka/kafka_2.12-2.4.1$ bin/kafka-console-producer.sh --broker-list localhost:9092 --topic Demo
>kafka is awesome
>hello everyone
>One
>Two
>Three
>kafka with multiple consumers
>
```

Below the main terminal area, there are two smaller terminal windows. The left one shows the output of the producer command:

```
localhost:9092 --topic Demo --from-beginning
kafka is awesome
hello everyone
One
Two
Three
kafka with multiple consumers
```

The right one shows the output of the consumer command:

```
dhoop@dhoop-ip510:~/Downloads/kafka/kafka_2.12-2.4.1$ bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic Demo --from-beginning
kafka is awesome
hello everyone
One
Two
Three
kafka with multiple consumers
```

The status bar at the bottom indicates the terminal is running 'B:leva*' and the window title is 'dhoop-ip510' 12:06 06-Apr-20.

[View Full Size Image](#)

[View Full Size Image \(write a file to Kafka Producer \)](#)